CMSC 143: Introduction to Object-Oriented Programming with Robots

Lab 9: Operating Overloading
Due Monday, November 7, 2011

In this lab, we will flush out the *Fraction* we started in lecture. The Fraction class overloads specially named methods allowing it to emulate numeric types.[1]. Your class should be properly documented (each class and method should have a pydoc string).

Test Cases

Write a function `test()` that exercises the Fraction class. Before you add functionality you should write a small test for it. This test function should give you confidence the operators are implemented correctly.

```
def test():
    f = Fraction(2, 3)
    g = Fraction(4, 8)
    print("f: ", f)
    print("g: ", g)
    print ("f + g: ", f+g)
    print ("f * g: ", f*g)
```

Arithmetic and Comparison Operators

Along with addition and multiplication your class should implement division (`__truediv__`) and subtraction (`__sub__`). You should also implement the comparision function (`__cmp__`) which Python uses for the comparison operations `<, <=, >, >=, ==, !=`. The comparision function returns -1, 0, 1 depending on whether the object is less than, equal, or greater than the other object passed as a parameter. Next, implement `__neg__`, a unary operator, which returns a negated version of the Fraction. Finally, you should implement `__float__` which is used when a user converts your Fraction to a float object.

Mixed Arithmetic

Although now you can create arbitrary arithmetic expressions using fractions (e.g. `(f + g)/(-h * i)`), expressions like `Fraction(1, 2) * 2` fail since the Fraction class assumes `other` is a Fraction. Improve those methods by checking to see if `other` is a Fraction, and if not, convert it to a Fraction. The `isinstance(object, Class)` function returns `True` if the type of `object` is `Class` and `False` otherwise.

You can test for Fraction-ness: `isinstance(other, Fraction)` or int-ness: `isinstance(other,int)`.

When Python comes upon a mixed expression like `2 * Fraction(1, 2)` instead of calling `__mul__` Python will call `__rmul__` because the Fraction is on the right side of the operator. Implement `__radd__`, `__rmul__`, `__rsub__`, and `__rtruediv__`. Why are `__radd__` and `__rmul__` different from `__rsub__` and `__rtruediv__`?

Deliverables

`cmsc143_lab9_LASTNAME_FIRSTNAME.py` – Your program.

Learning Objectives

- More Practice Creating Classes
- Overload Operators
- Write Test Cases

[1] http://docs.python.org/reference/datamodel.html#emulating-numeric-types

```
def gcd(a, b):
    ''' Euclid's algorithm for greatest common denominator'''
    if b == 0:
        return a
    else:
        return gcd(b, a % b)

class Fraction:
    ''' A user-defined fraction class for exact rational numbers '''

    def __init__(self, num, denom):
        ''' Creates a new Fraction object num/denom'''
        self.num = num
        self.denom = denom
        self.reduce()

    def __repr__(self):
        ''' returns string representation of our fraction'''
        return str(self.num) + "/" + str(self.denom)

    def reduce(self):
        ''' converts our fractional representation into reduced form'''
        divisor = gcd(self.num, self.denom)
        self.num = self.num // divisor
        self.denom = self.denom // divisor

    def __mul__(self, other):
        '''return a new fraction that is the result of multiplying (*) this fraction by other'''
        newnum = self.num * other.num
        newdenom = self.denom * other.denom
        return Fraction(newnum, newdenom)

    def __add__(self, other):
        '''return a new fraction that is the result of adding (+) this fraction by other'''
        newnum = self.num * other.denom + self.denom*other.num
        newdenom = self.denom * other.denom
        return Fraction(newnum, newdenom)

    def __truediv__(self, other):
        pass

    def __sub__(self, other):
        pass

    def __cmp__(self, other):
        pass

    def __neg__(self):
        pass

    def __float__(self):
        pass
```